Rusty swapping:
rewriting a zswap backend in Rust

Vitaly Wool, Konsulko AB

About me: Vitaly

Konsulko Group

- Has been working with embedded Linux since 2003
- Currently living in Malmö, Sweden
- Staff Engineer at Konsulko Group
- Managing Director at Konsulko AB

About this presentation

Konsulko Group

- Swapping and zswap
- Allocator backends for zswap
- zblock in a nutshell
- Kickstart coding in Rust
- z (rusty) block
- Comparisons
- Conclusions

(z)swapping

Swapping (paging)

Konsulko Group

- using secondary storage to store and retrieve data
  - secondary storage is usually an SDD or flash device
  - saves memory by pushing rarely used pages out
- trade memory for performance?
  - reading and writing pages may be quite slow
- use RAM to cache swapped–out pages
  - compress swapped–out pages, or there's no gain
- trade performance for memory?
  - in some sense, but now we are more flexible

zswap --
compressed write-back cache

Konsulko Group

- compresses swapped-out pages and moves them into a pool
  - when the pool is full enough, pushes the compressed pages to the secondary storage
  - pages are read back directly from the storage when needed
- compression is implemented using crypto API
  - several compression backends (lz4, lzo, gzip...)
- allocation is implemented using zpool API
  - 3 allocation backends (zbud, zsmalloc, z3fold)

Allocator backends

allocators: why and how?

Konsulko Group

- Special purpose allocators to work with small objects
  - typical object: a compressed page (len 0x0002..0x1000)
  - use zpool API
- zbud
  - up to 2 objects (buddies) per page
- zsmalloc
  - objects span across pages
- z3fold
  - up to 3 objects per page

zpool API

Konsulko Group

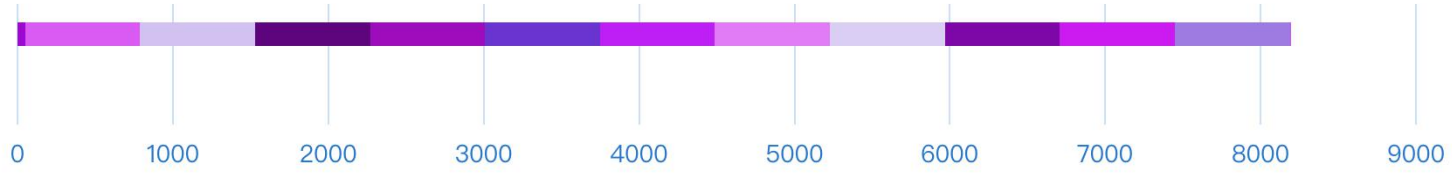
```
struct zpool_driver {
        char *type;
        struct module *owner;
        atomic_t refcount;
        struct list_head list;

        void *(*create)(const char *name, gfp_t gfp);
        void (*destroy)(void *pool);

        bool malloc_support_movable;
        int (*malloc)(void *pool, size_t size, gfp_t gfp,
                                unsigned long *handle);
        void (*free)(void *pool, unsigned long handle);

        bool sleep_mapped;
        void *(*map)(void *pool, unsigned long handle,
                                enum zpool_mapmode mm);
        void (*unmap)(void *pool, unsigned long handle);

        u64 (*total_size)(void *pool);
};
```

zblock in a nutshell

zblock: simple high density allocator

Konsulko Group

- we allocate blocks of pages for same size objects
  - `block = __get_free_pages(gfp, block_desc[block_type].order);`
- block descriptors are defined the following way:
  - ```
    #define SLOT_SIZE(nslots, order) \
            (round_down((BLOCK_DATA_SIZE(order) / nslots), sizeof(long)))
    ```
  - ```
    static const struct block_desc {
            const unsigned int slot_size;
            const unsigned short slots_per_block;
            const unsigned short order;
    } block_desc[] = {
    ...
            { SLOT_SIZE(11, 1), 11, 1 },
    ```
- 2 pages minus header = 8144 bytes
  - 8144 = 740*11 + 4, i. e. 11 chunks of 740 bytes



zblock in a nutshell

Konsulko Group

- Allocate:
  - find an appropriate *block_type* for a given object's size
  - go through the list of blocks of *block_type* to find an empty slot
    - allocate a new block if all the existing blocks are full
  - mark the slot as occupied and return the handle
- Free:
  - find the slot corresponding to the handle
  - mark the slot as free
  - free the whole block if there are no occupied slots now

zblock in comparison

Konsulko Group

- very simple
  - small code footprint
- highly configurable
- fast operation
- doesn't have 3x (z3fold) or 2x (zbud) ratio limit
- doesn't require MMU (unlike zsmalloc)
- compresison ratio beween z3fold and zsmalloc

Kickstarting coding in Rust

Why Rust?

Konsulko Group

- good
  - memory safety
  - variables immutable by default
- bad
  - immature kernel support
  - compilers are still not bug free
- subsystems
  - "*Rust should be deployed in subsystems, not drivers; USB stack and networking stack, first and foremost*" (Linus Walleij)

Environment is a mess

Konsulko Group

- a lot of requirements just to start building
  - you probably can't use rustc that comes with your Linux distro
  - most likely a very specific Rust version is needed to build your kernel
- check that the Rust setup is working:
  - make LLVM=1 rustavailable
- Documentation/rust/quick-start.rst is your friend

non-x86 support is incomplete

Konsulko Group

- Rust in kernel was mostly for x86
- ARM64 support has been added in 6.9
  - may be backported to earlier kernels with minimal trouble
  - still didn't work that well for Raspbian 6.6 kernel
- 32-bit ARM support is still lacking
  - timeline unclear

z (rusty) block

zblock: how do we rustify?

Konsulko Group

- zblock needs to call zpool_register_driver(↓)
  - C struct zpool_driver in Rust?
    - either use bindgen or do manually
    - bindgen produces nearly incomprehensible code
    - hard to mirror `struct list_head`

```
struct zpool_driver {
        char *type;
        struct module *owner;
        atomic_t refcount;
        struct list_head list;

        void *(*create)(const char *name, gfp_t gfp);
        void (*destroy)(void *pool);

        bool malloc_support_movable;
        int (*malloc)(void *pool, size_t size, gfp_t gfp,
                                unsigned long *handle);
        void (*free)(void *pool, unsigned long handle);

        bool sleep_mapped;
        void *(*map)(void *pool, unsigned long handle,
                                enum zpool_mapmode mm);
        void (*unmap)(void *pool, unsigned long handle);
```

```
#[repr(C)]
struct list_head {
    prev: *mut list_head,
    next: *mut list_head,
}

#[repr(C)]
pub struct zpool_driver {
    pub type_: &'static CStr,
    pub owner: *mut ZblockRust,
    pub list: list_head,
    pub create: fn(name: &'static CStr,
                   gfp: gfp_t) -> *mut ::core::ffi::c_void,
```

Rustify algorithms in zblock

Konsulko Group

- most of the algorithms are very simple
  - allocation according to the descriptor array
  - array search etc.
- array initialization is different
  - `memset()` won't work
  - have to learn Rust idioms

```
#[derive(Copy, Clone)]
struct BlockCache {
    c: [*mut ZblockBlock; BLOCK_CACHE_SIZE]
}

impl BlockCache {
    fn new() -> Self {
        Self { c: [ 0 as *mut ZblockBlock; BLOCK_CACHE_SIZE ] }
    }
}
```

Rustify spinlocks in zblock

Konsulko Group

- Rust spinlock mapping in kernel is nearly incomprehensible
  - arguably this is where safety was chosen over sanity
    - yes, you don't have to unlock
    - no, guards are not that convenient

```
fn cache_insert_block(block: *mut ZblockBlock, list: &mut BlockList)
{
    let mut guard = list.lock.lock();
    let mut min_index = 0;
    let mut min_free_slots = MAX_SLOTS;
    unsafe {
        for i in 0..BLOCK_CACHE_SIZE {
            if guard.c[i].is_null() || (*guard.c[i]).free_slots != 0 {
                min_index = i;
                break;
            }
```

Comparisons

module size

Konsulko Group

- arm64, unstripped

```
26712 Mar 24 17:46 mm/z3fold.o
 8512 Apr 15 08:23 mm/zblock.o
 6664 Mar 24 23:41 mm/zbud.o
 8040 Mar 24 23:41 mm/zpool.o
30464 Mar 24 23:41 mm/zsmalloc.o
42664 Mar 24 23:40 mm/zswap.o
```

- arm64, stripped

```
12760 Apr 17 08:45 mm/z3fold.o
 4040 Apr 17 08:45 mm/zblock.o
 3000 Apr 17 08:45 mm/zbud.o
 3432 Apr 17 08:45 mm/zpool.o
15056 Apr 17 08:46 mm/zsmalloc.o
16736 Apr 17 08:46 mm/zswap.o
```

- x86_64, unstripped

```
570128 Apr 17 08:37 mm/z3fold.o
215624 Apr 17 08:37 mm/zblock.o
424560 Apr 17 08:37 mm/zblock_rust.o
200600 Apr 17 08:37 mm/zbud.o
203528 Apr 17 08:37 mm/zpool.o
603264 Apr 17 08:37 mm/zsmalloc.o
645112 Apr 17 08:37 mm/zswap.o
```

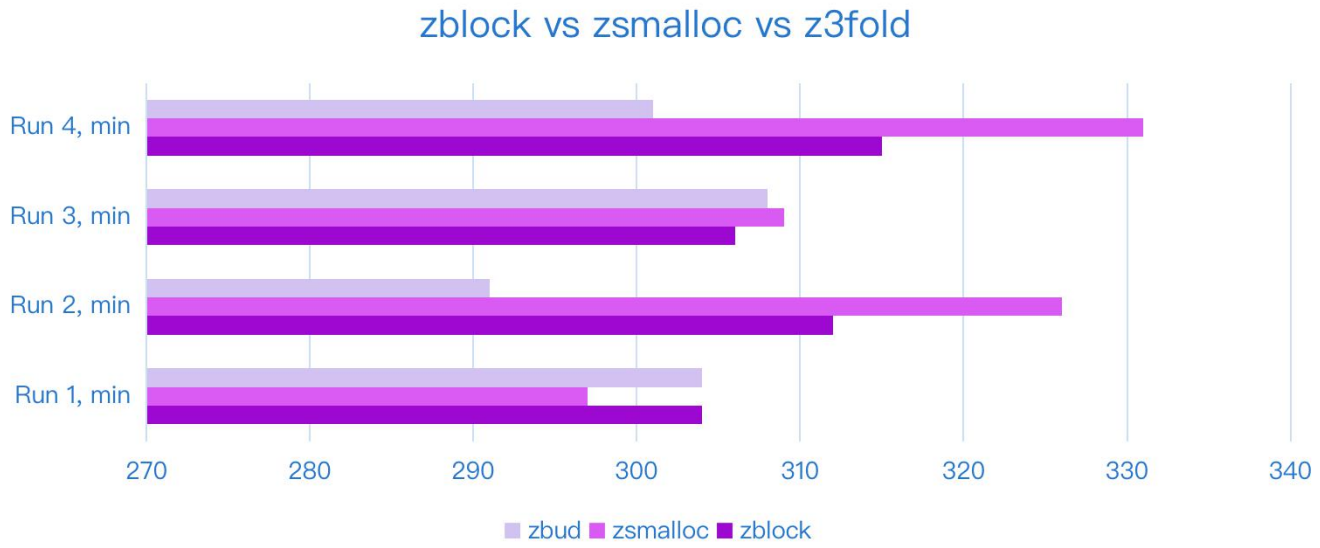
- x86_64, stripped

```
17656 Apr 17 08:50 mm/z3fold.o
 4784 Apr 17 08:50 mm/zblock.o
 3816 Apr 17 08:50 mm/zblock_rust.o
 3992 Apr 17 08:50 mm/zbud.o
 5392 Apr 17 08:50 mm/zpool.o
19288 Apr 17 08:50 mm/zsmalloc.o
23048 Apr 17 08:50 mm/zswap.o
```

Linux kernel compilation: zblock vs zsmalloc vs zbud

Konsulko Group

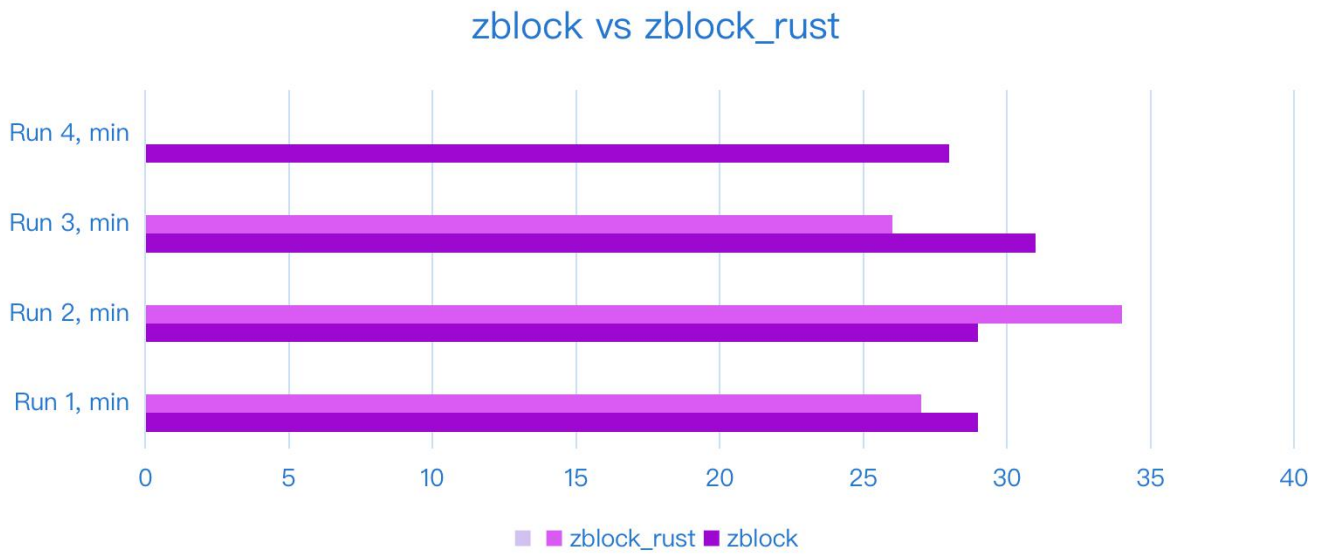
- tests run on Raspberry Pi 4B
  - make –j4
- zswap with LZ4



Linux kernel compilation: zblock vs zblock_rust

Konsulko Group

- tests run on QEMU x86_64
  - make –j4
- zswap with LZ4 and zblock or zblock_rust
  - zblock_rust hung once



Conclusions

Summarizing...

Konsulko Group

- Rust, you have a point
  - the code is executing fast
  - code footprint is small (may be even smaller that gcc's)
- It's not easy to get going
  - cumbersome environment setup for every kernel revision
- Real memory safety is questionable
  - many unsafe{} sections just have to be there
- Lacking support for important architectures
  - e. g. zblock would definitely target 32-bit ARM

Concluding...

Konsulko Group

- zblock is a zswap allocator backend trying to make its way into the mainline
- zblock is designed to be a replacement for z3fold
- zblock should work on low-end architectures
  - e. g. with small RAM footprint
  - e. g. 32-bit
  - e. g. without MMU
- 2 implementations (C, Rust) work equally well when comparable
- Rust implementation can't meet the goals above (yet?)

Thanks for your attention!

Vitaly Wool <vitaly.wool@konsulko.com>
Embedded OSS 2024, Seattle

“I'm gonna break my rusty cage... and run”